Deuxième partie :

Cours de Ière

Applications Delphi

7 Delphi

7.1 Introduction

Après son lancement, Delphi se présente sous la forme de 4 fenêtres.

La première fenêtre occupe la partie supérieure de l'écran. Elle correspond à l'environnement de programmation proprement dit.

Cette fenêtre contient :

- la barre de titre ;
- la barre de menu de Delphi ;
- une zone « barre d'outils » (sur la gauche) ;
- une zone contenant les divers composants regroupés par familles.

La seconde fenêtre se trouve par défaut à gauche de l'écran : c'est *l'inspecteur d'objets*. Il permet de visualiser, pour chaque objet ou composant, les propriétés et les événements auxquels l'objet peut répondre.

La troisième fenêtre constitue la fiche principale de la future application Delphi. Il s'agit, au départ, d'une fenêtre vide dans laquelle on placera les divers objets.

La dernière fenêtre, cachée sous la précédente constitue l’éditeur proprement dit, contenant le code source de l'application.

Pour démarrer une nouvelle application, il faut choisir l'option **New Application** du menu **File**.

Pour sauvegarder une application, il faut choisir l'option **Save All** du menu **File**. Une règle à suivre absolument est de créer un répertoire par application. Comme Delphi crée plusieurs fichiers pour une application donnée, il est plus facile de les retrouver s'ils ne sont pas enregistrés avec d'autres fichiers de noms pratiquement identiques.

Lors du premier « tout enregistrement » de l'application, une fenêtre permet de choisir l'emplacement de sauvegarde et même de le créer.

Pour exécuter une application, il faut choisir l'option **Run** du menu **Run**. Si les options d'auto-enregistrement ont été sélectionnées et que l'application n'a encore jamais été sauvegardée, la fenêtre d'enregistrement s'affiche. L'application est ensuite compilée puis exécutée, si elle ne contient pas d'erreur.

7.2 Les fichiers utilisés en Delphi

Les fichiers d'un projet :

.DPR	**fichier projet**	***Delphi Project File***
.DFM	**fichier fiche**	***Delphi Form File***
.PAS	**fichier unité - code source**	
.EXE	fichier exécutable (le programme développé)	
.DCU	fichier unité - code compilé	*Delphi Compiled Unit*
.RES	fichier ressource (icônes, bitmaps, curseurs, . . .)	
.DPL	fichier paquet compilé	*Delphi Package Library*
.DPK	fichier paquet source	*Delphi Package*

Les fichiers .DPR, .DFM et .PAS sont les fichiers nécessaires à la programmation et doivent être copiés pour continuer le développement sur une autre machine.

Autres fichiers :

.DOF	options du projet	*Delphi Options File*
.DSK	paramètres du bureau	*Delphi Desktop File*
.~??	fichiers de sauvegarde	

7.3 L’approche Orientée-Objet

Dans la programmation en Delphi, nous allons manipuler des objets. Ces objets sont définis par leurs propriétés, leurs méthodes et leurs événements.

Dans la vie courante, un objet peut être toute chose vivante ou non (par exemple : une voiture, une montre, …). En informatique, un objet est souvent un bouton, une fenêtre, un menu, …

7.3.1 Les propriétés

Cependant, chaque personne « voit » l’objet différemment. Par exemple chacun aura une perception différente de l’objet voiture, selon l’importance qu’il attribue aux caractéristiques de l’objet.

Une propriété est une information décrivant une caractéristique de l’objet.

Ainsi, il est facile d’énumérer quelques propriétés pour l’objet voiture : vitesse maximale, cylindrée, marque, modèle, couleur, …

Nous pouvons consulter les propriétés et également les modifier.

Par exemple, nous pouvons définir les propriétés d’une voiture dans un jeu de course, tel que la couleur. Ceci se fait de la manière suivante :

Voiture1.Couleur := Rouge

Bien entendu, il faut que la constante « Rouge » soit définie.

Les objets (dans Delphi ces objets sont appelés *composants*) que nous allons utiliser sont prédéfinis (boutons, fenêtres, menus, …). Pour afficher les propriétés d’un objet, il suffit de cliquer dessus. Les propriétés s’affichent alors dans l’inspecteur d’objet.

Il existe des composants en lecture seule.

7.3.2 Les méthodes

Pour simplifier, on peut se représenter une méthode comme un ordre du style « fais ceci ». Cet ordre provoque l’exécution d’une certaine action par l’objet.

Par exemple, pour l’objet voiture, on peut énumérer les méthodes suivantes : accélérer, freiner, changer de vitesse, …

Donc, l’instruction

Voiture1.Accélérer(10)

indique à la voiture qu’elle doit accélérer d’un facteur 10.

Les propriétés ne font que changer une caractéristique d’un objet alors que les méthodes effectuent une action. On n’utilise pas de signe d’affectation lorsqu’on exécute une méthode.

7.3.3 Les événements

Pour chaque objet, il peut survenir certains événements, qui déclenchent des réactions.

Dans l'exemple de la voiture, lorsqu'on tourne la clé dans le contact ou lorsqu'on appuie sur l'accélérateur, la voiture respectivement démarre ou accélère.

Pour les objets informatiques il leur arrive des événements auxquels ils peuvent réagir.

Par exemple, un bouton peut avoir les événements *OnMouse*… (événements liés à la souris), *OnKey*… (événements liés au clavier), *OnEnter* (réception du focus), *On Exit* (perte du focus), …

Les événements existants pour un objet sont visibles dans l'inspecteur d'objet.

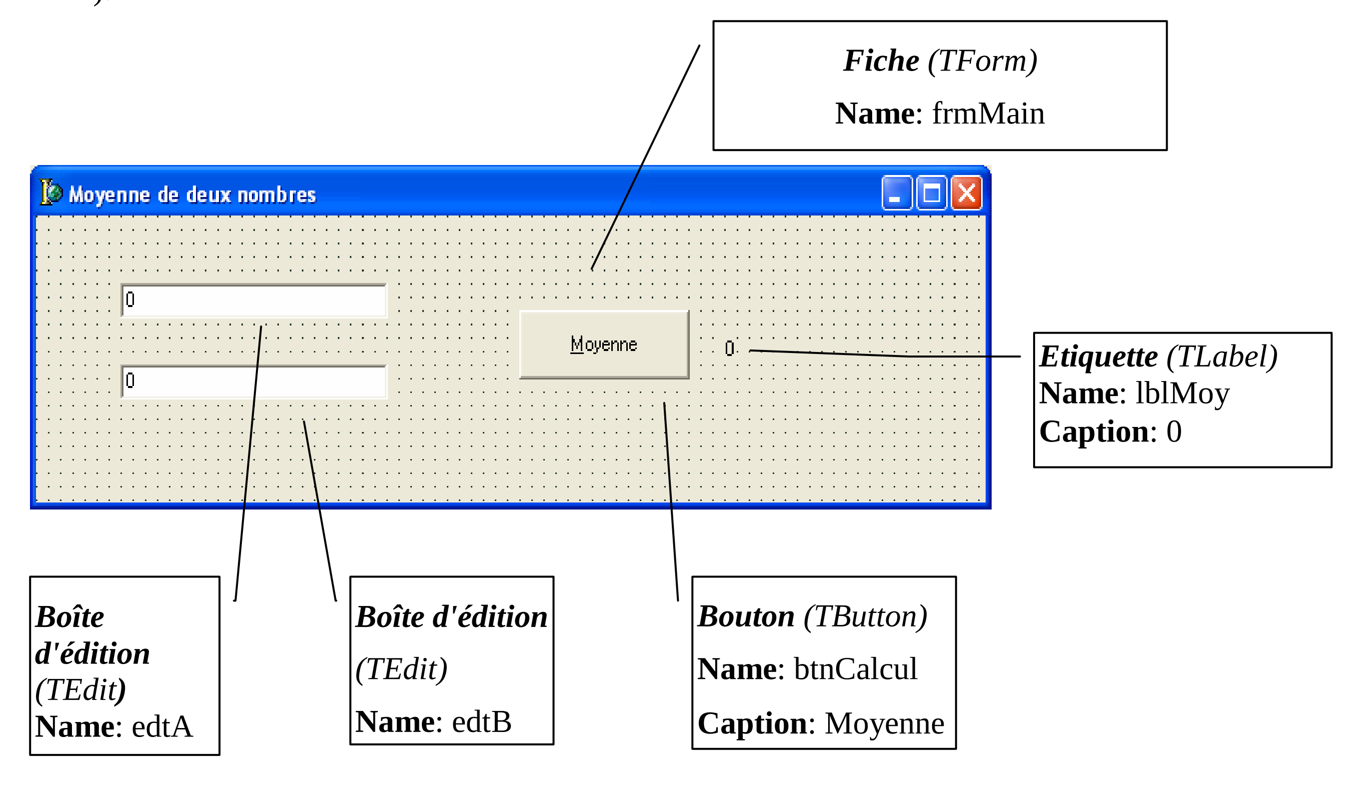
7.4 Passage Pascal – Delphi – un premier exemple

7.4.1 L'interface

En Delphi, nous utiliserons les composants de l'interface graphique (les fenêtres de Windows) pour entrer les données et afficher les résultats. Les algorithmes PASCAL que nous avons utilisés jusqu'à maintenant pour obtenir les résultats pourront rester inchangés.

D'abord, nous allons créer l'interface du programme. Nous allons adapter les noms internes (propriété **Name**) de chaque composant que nous utilisons.

En plus, nous allons modifier les inscriptions sur les différents composants (propriétés **Caption** ou **Text**).



7.4.2 Les conversions de types

Les données inscrites dans les boîtes d'édition sont de type texte (string). Nous devons donc les transformer afin de pouvoir effectuer des calculs.

Voici quelques fonctions permettant d'effectuer certaines conversions :

`StrToInt(string)` : convertit une chaîne de caractères en un nombre entier (type integer)

`StrToFloat(string)` : convertit une chaîne de caractères en un nombre réel (type real).

De même, pour pouvoir afficher le résultat, nous devons le transformer en texte. Ceci peut se faire grâce aux fonctions `FloatToStr` et `IntToStr`.

7.4.3 Le traitement

Après la saisie des données dans les boîtes d'édition, l'utilisateur va cliquer sur le bouton `btnCalcul`. À cet instant l'événement `OnClick` du bouton est généré et la méthode `btnCalculClick` est lancée. Nous allons donc entrer les instructions à effectuer dans la méthode `btnCalculClick` :

```
procedure TfrmMain.btnCalculClick(Sender: TObject);
var A,B,MOY : real;
begin
  A := StrToFloat(edtA.Text);
  B := StrToFloat(edtB.Text);
  MOY := (A+B)/2;
  lblMoy.Caption := FloatToStr(MOY);
end;
```

7.4.4 Exercices

Exercice 7-1

Ecrivez un programme qui affiche le plus grand de trois nombres réels A, B, C.

Exercice 7-2

Ecrivez un programme qui calcule la somme d'une série de nombres entrés au clavier, en utilisant deux boîtes d'édition et un bouton pour la remise à zéro de la somme.

Exercice 7-3

Réalisez le programme PUISSANCE qui calcule et affiche la puissance $\mathbf{X^N}$ (puissance X exposant N pour un réel X et un entier N positif, négatif ou zéro).

Pour les cas où $\mathbf{X^N}$ ne se laisse pas calculer, affichez un message d'erreur !

Exercice 7-4

a) Réalisez un programme qui permet de simplifier une fraction.

b) Utilisez une partie du programme réalisé sous a) pour faire un programme qui additionne deux fractions.

7.5 Calcul de la factorielle

Comme premier programme essayons d'implémenter en Delphi le calcul de la factorielle.

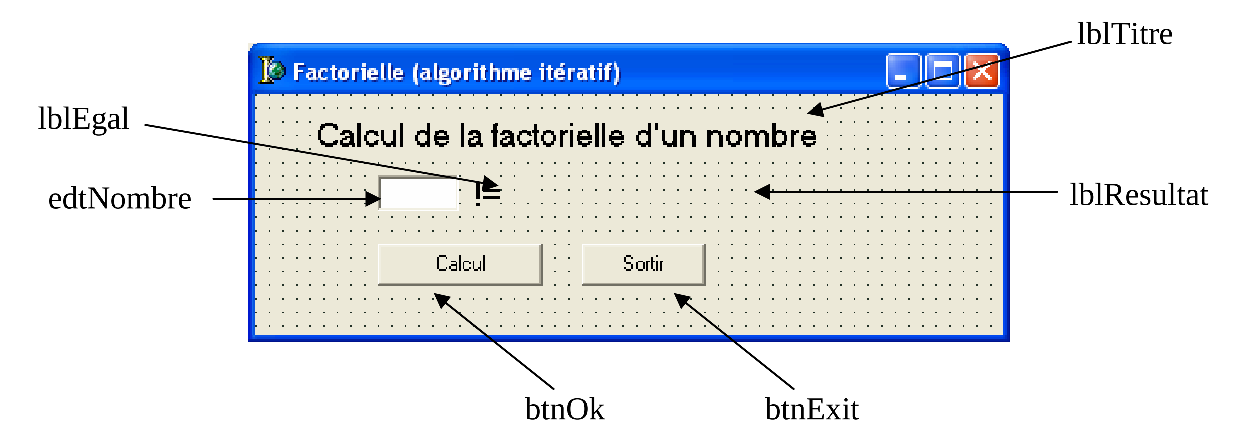
Rappelons que pour tout entier naturel *x*, $x! = \begin{cases} 1 \cdot \ldots \cdot x & \text{si } x \geq 1 \\ 1 & \text{si } x = 0 \end{cases}$

Pour élaborer ce calcul nous pouvons utiliser le programme développé dans le cours de 2[e] et l'incorporer dans celui en Delphi.

7.5.1 Présentation visuelle

Commençons par établir un formulaire dans lequel nous notons les valeurs et éditons les résultats.

Voici un exemple d'un tel formulaire.



Ce formulaire est un nouvel objet que nous appellerons *Tformulaire*, de capture (Caption) : **Factorielle (algorithme itératif)**. Il est composé des propriétés suivantes :

Name	type	text	Caption
lblTitre	TLabel		Calcul de la factorielle d'un nombre
lblEgal	TLabel		! =
edtNombre	TEdit	*valeur de la factorielle à calculer*	
lblResultat	TLabel		
btnOk	TButton		Calcul
btnExit	TButton		Sortir

Il est évident que tous ces champs possèdent encore davantage de propriétés. Nous n'avons énuméré ici que les plus importantes.

7.5.2 Code

Une fois ce formulaire établi, nous pouvons écrire le code nécessaire pour calculer la factorielle. Rappelons que nous y utiliserons le code *Pascal* établi en 2e (voir également les « Algorithmes obligatoires »).

Delphi s'occupera de la déclaration du formulaire et de ses propriétés.

La seule partie du code que nous devons écrire est celle de la procédure `btnOkClick` qui va être exécutée, comme son nom le dit, après que l'utilisateur ait poussé sur le bouton `Ok`. Nous dirons que la procédure s'exécute après l'événement `onClick` appliqué à la propriété `btnOk`.

Le tout se trouvera dans l'unité `Unit1`.

Voici une possibilité de code pour la procédure en question.

```
procedure Tformulaire.btnOkClick(Sender: TObject);
var n,fact:integer;
begin
  n:=StrToInt(edtnum.Text);
  fact:=factorielle(n);
  lblresult.Caption:=IntToStr(fact)
end;
```

Bien entendu, cette procédure suppose que la fonction factorielle(n:integer):integer est définie.

7.5.3 Explication du programme.

En regardant de près ce code quelques remarques s'imposent :

- Comme la procédure s'emploie dans le formulaire `Tformulaire`, elle s'appellera sous son nom complet : `Tformulaire.btnOkClick`.
- La valeur saisie du nombre est la valeur de la propriété `Text` du champ `edtNombre`. Nous notons donc cette valeur par `edtNombre.Text`. De plus, comme il s'agit d'une chaîne de caractères, nous devons encore transformer cette chaîne en une valeur numérique par la fonction `StrToInt`, fonction prédéfinie dans Delphi.
- La valeur de la factorielle calculée sera affectée à la propriété `Caption` du champ `lblResultat` que nous noterons par `lblResultat.Caption`. Comme de plus cette valeur doit être du type chaîne de caractères, nous devons transformer `fact` par la fonction `IntToStr`, autre fonction prédéfinie dans Delphi.

L'unité `Unit1` se présentera finalement ainsi :

```
unit Unit1;
interface
uses
Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls;

type
  Tformulaire = class(TForm)
    lblTitre: TLabel;
```

```
    edtNombre: TEdit;
    lblEgal: TLabel;
    btnOk: TButton;
    lblResultat: TLabel;
    procedure btnOkClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  formul: Tformulaire;
implementation
{$R *.dfm}
//fonction permettant de calculer une factorielle
function factorielle(n:integer):integer;
var fact:integer;
begin
  fact:=1;
  while n>1 do
  begin
    fact:=fact*n;
    n:=n-1
  end;
  result:=fact
end;

procedure TFormulaire.btnOKClick(Sender: TObject);
var n,fact:integer;
begin
  n:=StrToInt(edtnum.Text);
  fact:=factorielle(n);
  lblresult.Caption:=IntToStr(fact)
end;

procedure TFormulaire.btnexitClick(Sender: TObject);
begin
  Close;
end;

end.
```

7.5.4 Exécution du programme

Une fois ce code saisi, nous sauvegardons le tout dans un répertoire réservé à cette application. Nous cliquons ensuite sur le bouton qui représente un petit triangle vert et le programme s’exécutera.

7.5.5 Remarques

- La méthode `Tformulaire.btnExitClick` aura comme seule commande `Application.Terminate`. De cette manière l'événement `Click` lié au bouton `btnExit` aura comme effet net d'arrêter l'application.
- La saisie fautive respectivement d'un nombre négatif ou décimal ne conduira pas à un message d'erreur de la part du programme mais nous affichera un résultat erroné. Nous laissons au lecteur le soin de corriger le programme pour l'améliorer de ce point de vue.

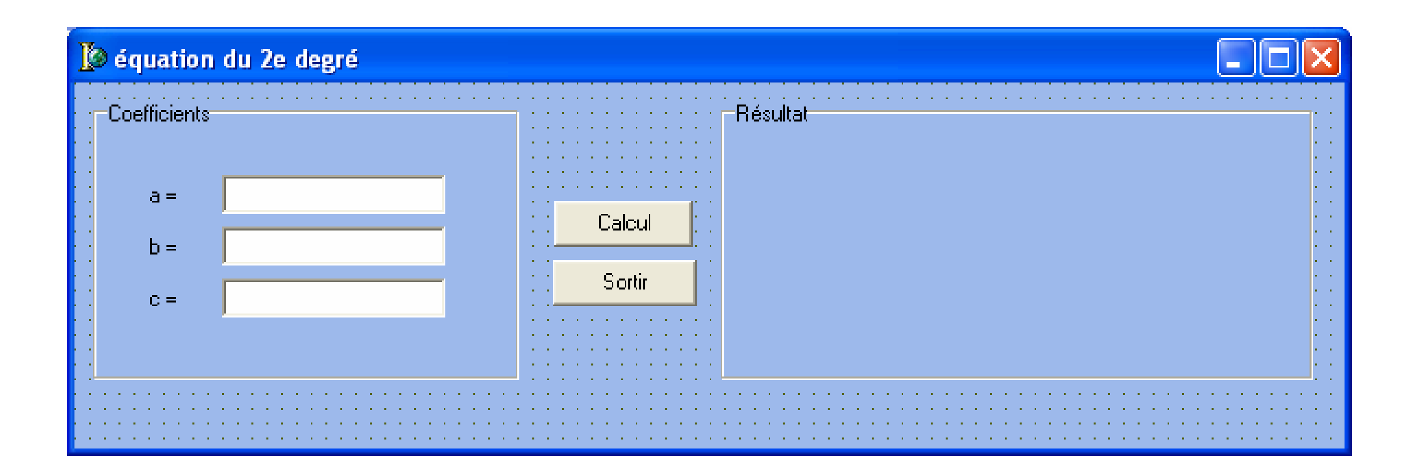
7.6 Equation du second degré

Écrivons maintenant un programme qui demande à la saisie les trois coefficients *a*, *b* et *c* d'une équation du second degré et qui calcule, si elles existent, les racines de l'équation $ax^2 + bx + c = 0$.

Ce même programme a été demandé comme exercice dans le cours de 2e. Nous en faisons ici un programme Delphi.

7.6.1 Présentation visuelle

Comme dans l'exemple précédent nous commençons par dessiner un formulaire que nous appellerons `Tformulaire`, dont l'instance `formulaire` nous permet de saisir les coefficients et de lire le(s) résultat(s). La propriété `Caption` de l'objet `Tformulaire` aura comme valeur : **équation du 2e degré**.



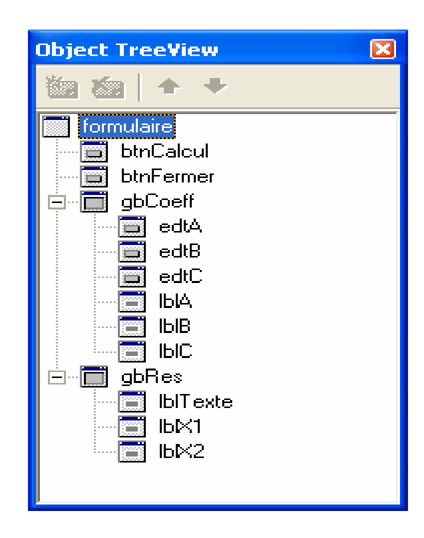
Ce formulaire est composé des champs suivants :

name	**type**	**text**	**caption**
lblA	TLabel		a =
lblB	TLabel		b =
lblC	TLabel		c =
lblTexte	TLabel		*texte sur le résultat*
lblX1	TLabel		*valeur de la racine*
lblX2	TLabel		*valeur de la racine*
edtA	TEdit	*valeur de a*	

edtB	TEdit	*valeur de b*	
edtC	TEdit	*valeur de c*	
btnCalcul	TButton		Calcul
gbCoeff	TGroupBox		Coefficients
gbRes	TGroupBox		Résultat
btnExit	TButton		Sortir

Nous remarquons tout-de-suite une nouvelle notion :

- les champs du type `TGroupBox` : Ils servent à regrouper différents champs dans un même groupe qu'ils affichent avec un cadre et un nom donné sous *Caption*. Dans notre exemple la TGroupBox `gbCoeff` regroupe les champs `lblA`, `lblB` et `lblC`, tandis la TGroupBox `gbRes` affichera les résultats et contient ainsi les champs `lblTexte`, `lblX1` et `lblX2`.
  Nous définissons ces `TGroupBox` comme suit :



- Nous venons déjà de remarquer que le résultat sera affiché dans la TGroupBox `gbRes`. Mais les étiquettes (labels) `lblTexte`, `lblX1` et `lblX2` sont invisibles pour l'instant.
  Si nous avons un résultat à afficher, `lblTexte` contiendra une des phrases suivantes :
  Il n'y a pas de solution réelle ! ,

  Il existe une solution réelle ! ou

  Il y a deux solutions réelles différentes ! (ceci en fonction du résultat du calcul), tandis que `lblX1` et `lblX2` contiendront les valeurs des racines éventuelles. Comme actuellement ces étiquettes ne contiennent pas de texte (*caption vide*), elles n'apparaîtront pas à l'écran.

7.6.2 Code

Une fois ce formulaire établi nous écrivons le code suivant :

```
unit Unit1;
interface
uses  Windows,  Messages,  SysUtils,  Classes,  Graphics,  Controls,
Forms, Dialogs, StdCtrls, Buttons;
type
  Tformulaire = class(TForm)
    gbCoeff: TGroupBox;
    lblA: TLabel;
    lblB: TLabel;
    lblC: TLabel;
    edtA: TEdit;
    edtB: TEdit;
    edtC: TEdit;
    btnCalcul: TButton;
    gbRes: TGroupBox;
    lblTexte: TLabel;
    lblX1: TLabel;
    lblX2: TLabel;
    btnExit: TButton;
    procedure btnCalculClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
  private
    { Private-Declarations}
  public
    { Public-Declarations }
  end; //Tformulaire
var formulaire: Tformulaire;
implementation
{$R *.DFM}
procedure Tformulaire.btnCalculClick(Sender: TObject);
var a,b,c,disc : real;
begin
  a:=StrtoFloat(edta.Text);
  b:=StrtoFloat(edtb.Text);
  c:=StrtoFloat(edtc.Text);
  disc:=b*b-4*a*c;
  if disc < 0 then
  begin
    lblTexte.Caption:='Il n''y a pas de solution réelle !';
    lblX1.Caption:='';
    lblX2.Caption:='';
  end
  else if round(disc*1000000) = 0 then  //un nombre reel n’est jamais 0
  begin
    lblTexte.Caption:='Il existe une solution réelle !';
    lblX1.Caption:='x = ' + FloattoStr(-b/(2*a));
    lblX2.Caption:='';
  end
  else if disc > 0 then
  begin
    lblTexte.Caption:='Il y a deux solutions réelles différentes !';
    lblX1.Caption:='x1 = ' + FloattoStr((-b-sqrt(disc))/(2*a));
```

```
    lblX2.Caption:='x2 = ' + FloattoStr((-b+sqrt(disc))/(2*a));
  end;
end;

procedure Tformulaire.btnExitClick(Sender: TObject);
begin
  Close;
end;

end.
```

7.6.3 Explications du programme

Nous lisons d'abord les 3 coefficients *a*, *b* et *c* de l'équation. Comme nous les avons définis comme variables réelles, nous devons utiliser la fonction `StrtoFloat` pour faire la transformation entre la chaîne de caractères que représente `edt*.Ttext` et les variables `a`, `b` et `c`.

Nous calculons ensuite le discriminant. En fonction du signe du discriminant nous envisageons les 3 cas :

- `disc<0` : il n'y a pas de résultat réel ;
- `disc=0` : il y a une racine ;
- `disc>0` : il y a deux racines réelles distinctes.

En fonction des différents cas nous calculons les valeurs des racines.

À la fin il nous reste encore à transformer les valeurs réelles, résultats des calculs, en chaînes de caractères pour les affecter à la propriété `Caption` des différentes étiquettes. Nous faisons ceci avec la fonction `FloatToStr`.

La méthode `Tformulaire.btnExitClick` aura comme seule commande `Application.Terminate`. De cette manière l'événement `Click` lié au bouton `btnExit` aura comme effet-net d'arrêter l'application.

7.7 Vérification du numéro de matricule

Développons ici un exercice qui prend en entrée le numéro de matricule d'une personne et qui vérifie que le chiffre de contrôle est correct.

7.7.1 Rappelons la méthode de calcul du numéro de contrôle.

Si nous notons $a_1 a_2 a_3 a_4 m_1 m_2 j_1 j_2 n_1 n_2 c$ un numéro de matricule, nous formons le dernier chiffre en parcourant les étapes suivantes :

- Nous formons la somme :
  $sum = 5*a_1 + 4*a_2 + 3*a_3 + 2*a_4 + 7*m_1 + 6*m_2 + 5*j_1 + 4*j_2 + 3*n_1 + 2*n_2$;
- soit *n* le reste de la division de *sum* par 11 ;
- si *n=1* il y a une faute ;
- si *n=0* le chiffre de contrôle reste *0 ;*
- si $n \neq 0$ et $n \neq 1$ alors le chiffre de contrôle vaut $11 - n$.

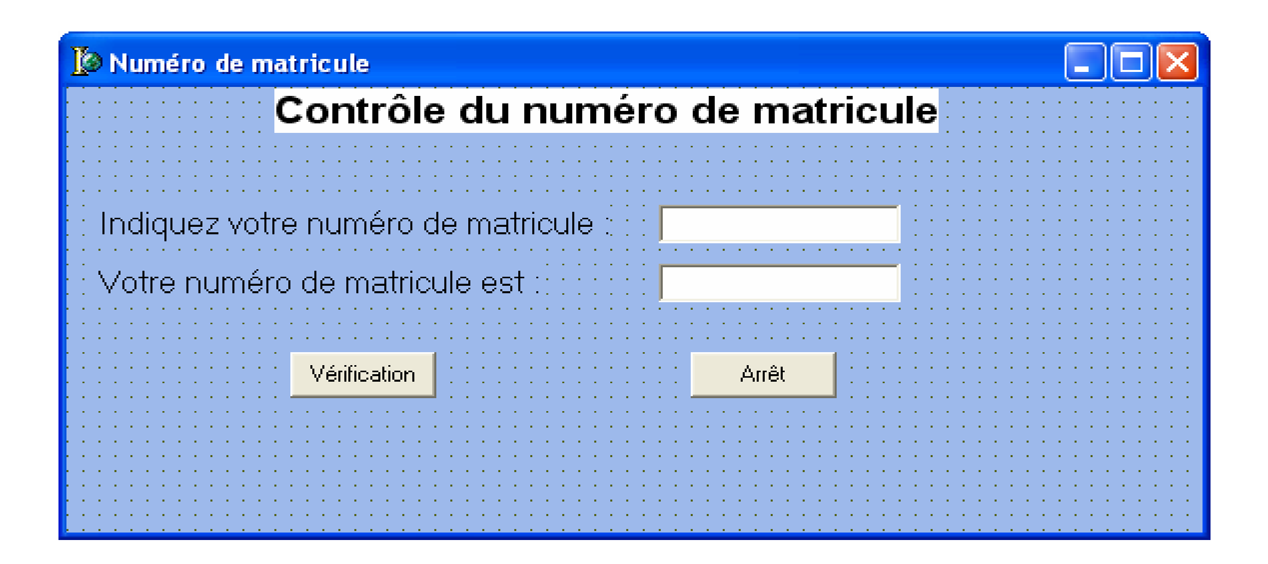
Nous essayons de traduire ceci en Delphi.

7.7.2 Présentation visuelle

Pour cela il nous faut d'abord un formulaire que nous appellerons, comme toujours, `Tformulaire` dont une instance nous servira à manipuler les entrées et sorties.

La valeur de la propriété `Caption` de l'objet `Tformulaire` sera : Numéro de matricule.

Voici un exemple d'un tel formulaire.



Il contient les éléments suivants :

nom	type	text	caption
lblTitre	TLabel		Contrôle du numéro de matricule
lblSaisie	TLabel		Indiquez votre numéro de matricule :
lblResultat	TLabel		Votre numéro de matricule est :
edtSaisie	TEdit	*numéro de matricule*	
edtResultat	TEdit	*correct/faux*	
btnVerif	TButton		Vérification
btnExit	TButton		Sortie

7.7.3 Code

Une fois ce formulaire établi, nous devons programmer le code nécessaire.

Voici un exemple d'implémentation.

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls;
type
  TfrmMatricule = class(TForm)
    lblTitre: TLabel;
    lblSaisie: TLabel;
    lblResultat: TLabel;
    edtSaisie: TEdit;
    edtResultat: TEdit;
    btnVerif: TButton;
    lblResultat: TLabel;
    btnExit: TButton;

    procedure btnVerifClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end; //Tformulaire
var
  frmMatricule: TfrmMatricule;

implementation
{$R *.dfm}
procedure TfrmMatricule.btnVerifClick(Sender: TObject);
var
  a1,a2,a3,a4,m1,m2,j1,j2,n1,n2,nc,c: integer;
  sum : integer;
  s : string;
begin
  s:=edtSaisie.Text;
  if length(s) <> 11 then
    ShowMessage('Numéro de matricule mal saisi')
  else
  begin
    a1:= StrToInt(copy(s,1,1));
    a2:= StrToInt(copy(s,2,1));
    a3:= StrToInt(copy(s,3,1));
    a4:= StrToInt(copy(s,4,1));
    m1:= StrToInt(copy(s,5,1));
    m2:= StrToInt(copy(s,6,1));
    j1:= StrToInt(copy(s,7,1));
```

```
    j2:= StrToInt(copy(s,8,1));
    n1:= StrToInt(copy(s,9,1));
    n2:= StrToInt(copy(s,10,1));
    nc:= StrToInt(copy(s,11,1));
    sum := 5*a1+4*a2+3*a3+2*a4+7*m1+6*m2+5*j1+4*j2+3*n1+2*n2;
    sum := sum mod 11;
    if sum = 1 then s:='faux'
    else
    begin
      if sum = 0 then c:= 0
      else c:= 11-sum;
      if nc=c then s:='correct'
      else s:='faux';
    end;
    edtResultat.Text:=s;
  end;
end;
procedure TfrmMatricule.btnExitClick(Sender: TObject);
begin
  Close;
end;
end.
```

7.7.4 Explication du code

La variable `s` va contenir le numéro de matricule saisi. C'est la valeur saisie.

Pour éviter qu'un utilisateur ne donne qu'une partie d'un numéro de matricule, nous faisons un test sur la longueur du numéro et nous affichons une erreur si la longueur ne correspond pas.

Le message d'erreur est affiché par la procédure `ShowMessage` dont la syntaxe est la suivante :

```
ShowMessage(msg: string);
```

où

msg chaîne de caractères à afficher.

Ensuite nous extrayons les différentes valeurs du numéro de matricule. Comme toutes les valeurs saisies sont des caractères nous devons les transformer en entiers par la fonction `StrToInt`.

La fonction `copy` sert à extraire des parties de chaînes de caractères. Sa syntaxe est la suivante :

```
Copy(s, index, count: Integer): string;
```

où :

S	chaîne de laquelle est extraite la partie ;
Index	indice de début de la chaîne à extraire (commence par 1) ;
Count	nombre de caractères à extraire.

Les étapes suivantes correspondent à l'algorithme énoncé.

Pour terminer nous éditons le résultat comme texte du champ `edtResultat`.

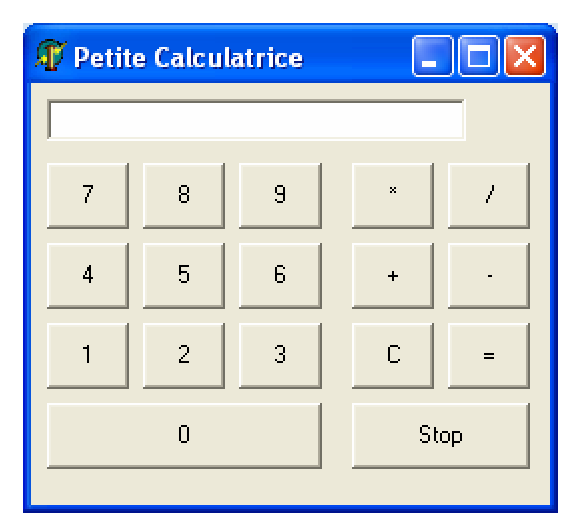
La procédure `Tformulaire.btnExitClick` sert de nouveau à arrêter l'application.

7.8 Une petite machine à calculer

Dans ce prochain exercice nous nous proposons de mettre en œuvre une petite machine à calculer, qui effectuera les 4 opérations élémentaires.

7.8.1 Présentation visuelle

Comme toujours il nous faut définir d'abord un formulaire. Voici une possibilité d'une telle interface entre l'opérateur et la machine.



Nous appellerons, comme toujours Tformulaire l'objet que nous allons définir ci-dessous.

Nom	type	text	caption
edtNum	TEdit	Saisie des nombres et des opérateurs qui interviennent dans le calcul	
btnButton0	TButton		0
btnButton1	TButton		1
btnButton2	TButton		2
btnButton3	TButton		3
btnButton4	TButton		4
btnButton5	TButton		5
btnButton6	TButton		6
btnButton7	TButton		7
btnButton8	TButton		8
btnButton9	TButton		9
btnButtonclear	TButton		C
btnButtondiv	TButton		/
btnButtonequal	TButton		=
btnButtonminus	TButton		-

btnButtonmult	TButton		*
btnButtonplus	TButton		+
btnButtonarret	TButton		Stop

7.8.2 Code

Une possibilité de code pour cette machine à calculer est le suivant :

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms,Dialogs, StdCtrls;

type
  Tformulaire = class(TForm)
    edtNum: TEdit;
    btnButton7: TButton;
    btnButton1: TButton;
    btnButton9: TButton;
    btnButton8: TButton;
    btnButton6: TButton;
    btnButton5: TButton;
    btnButton2: TButton;
    btnButton4: TButton;
    btnButton3: TButton;
    btnButton0: TButton;
    btnButtonmult: TButton;
    btnButtondiv: TButton;
    btnButtonclear: TButton;
    bnButtonminus: TButton;
    btnButtonplus: TButton;
    btnButtonequal: TButton;
    btnButtonArret: TButton;
    procedure FormCreate(Sender: TObject);
    procedure btnButton0Click(Sender: TObject);
    procedure btnButtonplusClick(Sender: TObject);
    procedure bnButtonminusClick(Sender: TObject);
    procedure btnButtonmultClick(Sender: TObject);
    procedure btnButtondivClick(Sender: TObject);
    procedure btnButtonclearClick(Sender: TObject);
    procedure btnButtonequalClick(Sender: TObject);
    procedure btnButtonArretClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    flag, n1, n2, op : integer;
    n3 : real;
```

```
  end;

var
  formulaire: Tformulaire;

implementation

{$R *.dfm}

procedure Tformulaire.FormCreate(Sender: TObject);
begin
  flag:=1;
end;

procedure Tformulaire.btnButton0Click(Sender: TObject);
var strNum:string;
begin
  if Sender=btnButton0 then strNum:='0' else
  if Sender=btnButton1 then strNum:='1' else
  if Sender=btnButton2 then strNum:='2' else
  if Sender=btnButton3 then strNum:='3' else
  if Sender=btnButton4 then strNum:='4' else
  if Sender=btnButton5 then strNum:='5' else
  if Sender=btnButton6 then strNum:='6' else
  if Sender=btnButton7 then strNum:='7' else
  if Sender=btnButton8 then strNum:='8' else
  strNum:='9';

  if flag=0 then edtNum.Text:=edtNum.Text + strNum
  else
  begin
    edtNum.Text:=strNum;
    flag:=0
  end;
end;

procedure Tformulaire.btnButtonplusClick(Sender: TObject);
begin
  n1:=strtoint(edtNum.Text);
  flag := 1;
  op := 1;
end;

procedure Tformulaire.bnButtonminusClick(Sender: TObject);
begin
  n1:=strtoint(edtNum.Text);
  flag := 1;
  op := 2;
end;
```

```
procedure Tformulaire.btnButtonmultClick(Sender: TObject);
begin
  n1:=StrToInt(edtNum.Text);
  flag := 1;
  op := 3;
end;

procedure Tformulaire.btnButtondivClick(Sender: TObject);
begin
  n1:=StrToInt(edtNum.Text);
  flag := 1;
  op := 4;
end;

procedure Tformulaire.btnButtonclearClick(Sender: TObject);
begin
  edtNum.Text := '';
  flag := 1;
end;

procedure Tformulaire.btnButtonequalClick(Sender: TObject);
begin
  n2:=StrToInt(edtNum.Text);
  case op of
    1: n3:=n1+n2;
    2: n3:=n1-n2;
    3: n3:=n1*n2;
    4: n3:=n1/n2;
  end;//case
  edtNum.Text:=FloatToStr(n3);
  flag := 1;
end;

procedure Tformulaire.btnButtonArretClick(Sender: TObject);
begin
  Close;
end;
end.
```

7.8.3 Explication du code

Les méthodes invoquées par les boutons 0...9 de la calculatrice étant similaires, on peut se servir d’une seule procédure (au lieu de dix !!) pour réagir à l’actionnement des différentes touches numériques.

Cette procédure commune étant définie uniquement pour l’événement `btnButton0Click` « `procedure Tformulaire.btnButton0Click(Sender : TObject)` », les neuf autres touches numériques doivent donc produire le même événement. Pour cela, on choisira dans l’onglet « Events » de l’inspecteur d’objets la bonne procédure `btnButton0Click`.

Pour discerner la touche numérique qui a déclenché la procédure et ainsi donner la bonne valeur à la variable `strNum`, on compare le contenu de la variable `Sender` avec les noms des différentes touches numériques (`if Sender = btnButton ... then ...`).

Les mêmes variables `flag, op, n1, n2 et n3` sont utilisées dans toutes les procédures, et sont donc déclarées dans l'en-tête du programme (variables globales).

La variable `flag` est initialisée à 0 lors du lancement du formulaire (événement `FormCreate` du formulaire). Si `flag = 0`, le chiffre correspondant à la touche numérique actionnée est concaténé à la chaîne de caractères se trouvant déjà dans `edtNum.Text`.

Si par contre `flag = 1`, le chiffre correspondant à la touche numérique actionnée est copié dans `edtNum.Text` tout en écrasant le contenu antérieur.

En cliquant sur un signe d'opération (+ , - , * , /), le contenu de la propriété `Text` de l'objet `edtNum` est copié dans la variable `n1` et constitue le premier opérande. La variable `op` est initialisée avec le code correspondant à l'opération visée et la valeur 1 est assignée à la variable `flag`. Ainsi le chiffre suivant tapé sur les touches numériques de la calculatrice écrase le contenu de l'affichage (car `flag = 1`) et constitue le premier chiffre du deuxième opérande. La variable `flag` est alors remise à 0 et les chiffres suivants sont concaténés au deuxième opérande.

En tapant sur la touche (=), le contenu de la propriété `Text` de l'objet `edtNum` est copié dans la variable `n2` et consitue le deuxième opérande. L'opération définie par le code contenu dans la variable `op` est alors effectuée à l'aide de la structure alternative à choix multiples (instruction `case ... of ...`).

Il reste à remarquer que la calculatrice ne respecte pas la priorité des opérations.

7.9 Calcul matriciel - utilisation du composant StringGrid

Le prochain programme que nous allons établir est un programme qui manipule les opérations sur les matrices carrées 2x2.

Exercice :

Il est laissé au lecteur la possibilité de changer ce programme pour la manipulation des matrices carrées à 3 dimensions.

7.9.1 Le composant StringGrid

Dans cet exercice nous utilisons le type prédéfini : matrice ou `StringGrid` qui se trouve dans la barre des objets sous *Additional*. Il possède de nouvelles propriétés dont nous énumérons ici les plus importantes :

propriété	**Type**	**explications**
ColCount	Integer	nombre de colonnes
RowCount	Integer	nombre de lignes
FixedCols	Integer	nombre de colonnes d'en-têtes
FixedRows	Integer	nombre de lignes d'entêtes
DefaultColWidth	Integer	largeur des colonnes (pixels)
DefaultRowHeight	Integer	hauteur des colonnes (pixels)
Cells		ensemble de cellules
goEditing (Options)	Boolean	indique si l'utilisateur peut introduire des valeurs dans les cellules

Nous référençons une cellule par `Cells[colonne, ligne]`.

Attention : nous devons faire attention que le comptage des lignes et des colonnes commence, comme si souvent, par 0.

7.9.2 Le composant ListBox

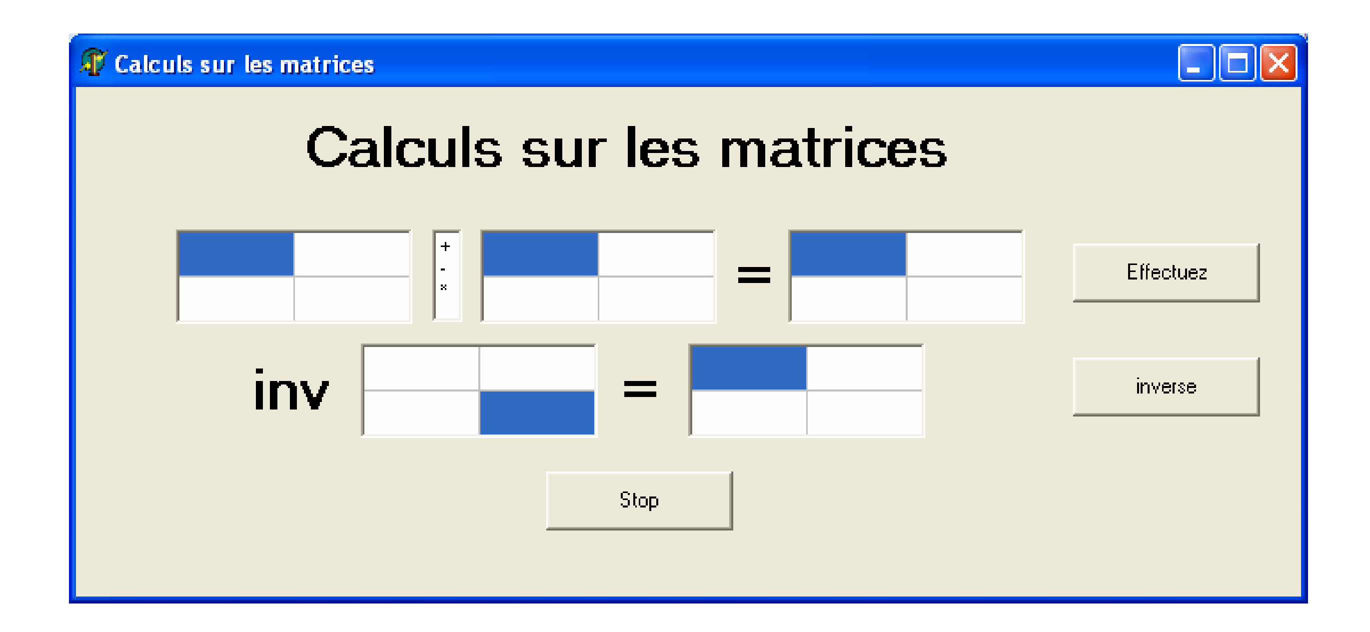
Dans cet exemple nous utilisons aussi un nouveau type, la `ListBox`. Elle sert à afficher plusieurs lignes de caractères et à donner la possibilité à l'utilisateur de choisir une ligne précise.

Nous trouvons la `ListBox` dans la barre des composants standards. Trois propriétés sont importantes à relever :

Propriété	**type**	**explications**
Items	string	tableau des lignes de la liste
ItemIndex	entier	indice de la ligne sélectionnée
Sorted	booléen	lignes triées ou non

7.9.3 Présentation visuelle

Commençons d'abord, comme dans les exercices précédents, par établir un formulaire qui nous sert à saisir les données et à les afficher. Voici un exemple d'un tel formulaire.



Ce formulaire présente les composants suivants :

name	**type**	**text**	**caption**
lblTitre	TLabel		Calculs sur les matrices
lblEg1	TLabel		=
lblEg2	TLabel		=

lblInv	TLabel		inv
btnEff	TButton		effectuez
btnInv	TButton		inverse
btnArret	TButton		Stop
name	**type**	**items**	
lbOp	TListBox	+ - *	
name	**type**	**colcount/rowcount**	**fixedcols/fixedrows**
sgMat1	TStringGrid	2/2	0/0
sgMat2	TStringGrid	2/2	0/0
sgMat3	TStringGrid	2/2	0/0
sgMatInv	TStringGrid	2/2	0/0
sgMatRes	TStringGrid	2/2	0/0

Les composants de type `StringGrid` qui servent à introduire des matrices doivent avoir l'option `goEditing` avec la valeur `True`.

Le champ `lbOp` de type `ListBox` sert à énumérer les différentes opérations et à donner à l'utilisateur la possibilité de choisir.

Après l'élaboration de ce formulaire nous pouvons écrire le code nécessaire. Voici un exemple possible.

```
unit Unit1 ;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls, Grids, Buttons ;

type
  Tformulaire = class(TForm)
    lblTitre : Tlabel ;
    sgMat1 : TstringGrid ;
    sgMat2 : TstringGrid ;
    sgMatRes : TstringGrid ;
    sgMat3 : TstringGrid ;
    sgMatInv : TstringGrid ;
    lblEg1: Tlabel;
    lblEg2: Tlabel;
    lblInv: Tlabel;
    lbOp: TlistBox;
    btnEff: Tbutton;
    btnInv: Tbutton;
    btnArret: Tbutton;
```

```
    procedure btnEffClick(Sender: Tobject);
    procedure btnInvClick(Sender: Tobject);
    procedure btnArretClick(Sender: Tobject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;//Tformulaire

var
  formulaire: Tformulaire;

implementation
{$R *.dfm}
procedure Tformulaire.btnEffClick(Sender: TObject);
var a,i,j : integer;
begin
  if lbOp.ItemIndex=0 then
    for i:=0 to 1 do
      for j:=0 to 1 do
      begin
        a:=StrToInt(sgMat1.Cells[i,j])+StrToInt(sgMat2.Cells[i,j]);
        sgMatRes.Cells[i,j]:=IntToStr(a);
      end;
      //j
    //i
  //fi index=0
  if lbOp.ItemIndex=1 then
    for i:=0 to 1 do
      for j:=0 to 1 do
      begin
        a:= StrToInt(sgMat1.Cells[i,j])-StrToInt(sgMat2.Cells[i,j]);
        sgMatRes.Cells[i,j]:=IntToStr(a);
      end;
      //j
    //i
  //fi index=1
  if lbOp.ItemIndex=2 then
    for i:=0 to 1 do
      for j:=0 to 1 do
      begin
        a:=StrToInt(sgMat1.Cells[0,j])*StrToInt(sgMat2.Cells[i,0])
        +StrToInt(sgMat1.Cells[1,j])*StrToInt(sgMat2.Cells[i,1]);
        sgMatRes.Cells[i,j]:=IntToStr(a);
      end;
      //j
    //i
  //fi index=0
end;//btnEffClick

procedure Tformulaire.btnInvClick(Sender: TObject);
var a,det:real;
begin
```

```
  det:=StrToInt(sgMat3.Cells[0,0])*StrToInt(sgMat3.Cells[1,1])
     -StrToInt(sgMat3.Cells[1,0])*StrToInt(sgMat3.Cells[0,1]);
  a:= 1/det*StrToFloat(sgMat3.Cells[1,1]);
  sgMatInv.Cells[0,0]:=FloatToStr(a);
  a:= (-1/det)*StrToInt(sgMat3.Cells[1,0]);
  sgMatInv.Cells[1,0]:=FloatToStr(a);
  a:= (-1/det)*StrToInt(sgMat3.Cells[0,1]);
  sgMatInv.Cells[0,1]:=FloatToStr(a);
  a:= 1/det*StrToInt(sgMat3.Cells[0,0]);
  sgMatInv.Cells[1,1]:=FloatToStr(a);
end;//btnInvClick

procedure Tformulaire.btnArretClick(Sender: TObject);
begin
  Application.Terminate
end;//btnArret
end.//Unit1
```

7.9.4 Explication du code

Comme les opérations +, - et * exigent deux opérateurs, mais que l'opération *inverse* n'a besoin que d'un seul, nous avons pu regrouper les trois premiers sous une seule procédure.

La `TListBox lbOp` nous donne les valeurs suivantes :

ItemIndex	opération
0	addition
1	soustraction
2	multiplication

Pour le reste, le contenu des différentes parties des procédures correspond aux règles mathématiques qui définissent les opérations sur les matrices.

8 La récursivité

8.1 Exemple

La fonction suivante calcule une puissance de base réelle non nulle et d'exposant naturel :

```
function puissance(x:real;m:integer):real;
begin
  if m=0 then result:=1
  else result:=x*puissance(x,m-1)
end;
```

Cette fonction présente une grande différence par rapport à toutes les fonctions que nous avons définies précédemment. Dans la définition même on trouve déjà un appel à la fonction puissance. Il s'agit ici d'un mécanisme très puissant, présent dans tous les langages de programmation modernes : la récursivité. Le fonctionnement exact de ce mécanisme ainsi que les conditions d'utilisation seront étudiées en détail dans les paragraphes suivants. Remarquons cependant qu'il existe un lien étroit entre la récursivité en informatique et la récurrence en mathématique. La définition de la fonction puissance présentée ici est une transcription quasi directe des formules $\begin{cases} x^0 = 1 \\ x^m = x \cdot x^{m-1} \end{cases}$, valables pour x non nul[5].

8.2 Définition : « fonction ou procédure récursive »

On dit qu'une fonction ou une procédure est *récursive* (de manière directe) si elle s'appelle elle-même. Une fonction ou une procédure est *récursive de manière indirecte* si elle appelle une autre fonction ou procédure qui rappelle la première de façon directe ou indirecte.

La fonction puissance du paragraphe précédent est bien sûr une fonction récursive **directe**.

Le mécanisme de la récursivité est très puissant, mais il faut une certaine expérience pour pouvoir l'utiliser dans de bonnes conditions. Dans la suite nous allons élucider principalement les aspects suivants :

- Sous quelles conditions et pourquoi une fonction récursive donne-t-elle le résultat attendu ? Comment vérifier qu'une telle fonction est correcte ?
- Comment le système gère-t-il une fonction récursive ? C'est-à-dire comment est-ce que ce mécanisme fonctionne en pratique ?
- Est-ce qu'une fonction récursive est « meilleure » ou « moins bonne » qu'une fonction itérative (normale) ?

Il est clair que ces différents aspects ne sont pas indépendants les uns des autres, mais qu'il faut une vue d'ensemble pour bien les comprendre.

[5] La fonction «`puissance`» donne un résultat incorrect si x et m sont nuls. De plus, il est nécessaire que m soit un entier positif !

8.3 Etude détaillée d'un exemple

Dans ce paragraphe nous revenons à la fonction « puissance » de la page précédente et nous allons commencer par étudier quelques exemples d'exécution.

`puissance(7,0)` : donne bien sûr comme résultat 1 vu que la condition `m=0` est vérifiée.

`puissance(7,1)` : la condition `m=0` n'est pas vérifiée et la fonction calcule donc d'abord «`x*puissance(x,m-1)`» c'est-à-dire «`7*puissance(7,0)`» ce qui donne dans une deuxième étape «`7*1=7`».

`puissance(7,2)` : ici la condition m=0 n'est pas non plus vérifiée et la fonction calcule donc aussi d'abord «`x*puissance(x,m-1)`» c'est-à-dire «`7*puissance(7,1)`» . Suivent ensuite les deux étapes précédentes. A la fin de la troisième étape le résultat obtenu est «`7*puissance(7,1)=7*[7*puissance(7,0)]=7*7*1=49`».

Il est important de remarquer ici que le système refait chaque fois toutes les étapes et « ne se souvient pas » des appels de fonctions précédents. L'exécution de `puissance(7,12)` nécessite 13 passages dans la fonction : d'abord 12 appels récursifs et ensuite un dernier passage où `m=0`.

L'exemple `puissance(7,-2)` est particulièrement intéressant. La condition `m=0` n'est pas vérifiée et la fonction calcule donc «`x*puissance(x,m-1)`» c'est-à-dire «`7*puissance(7,-3)`». Ensuite elle va évaluer «`7*puissance(7,-4)`», «`7*puissance(7,-5)`», «`7*puissance(7,-6)`», etc. Il est clair que cet appel ne va certainement pas donner le résultat 1/49. Mais la situation est plus grave, l'exécution de la fonction ne va pas donner de faux résultat, mais cette exécution ne va pas se terminer[6] vu que la condition `m=0` ne sera plus jamais vérifiée.

Pour qu'une fonction ou une procédure récursive s'arrête, il est nécessaire que le code vérifie les conditions suivantes :

- **Pour une ou plusieurs valeurs des données, appelées « cas de base », la fonction calcule directement (sans appel récursif) le résultat.**
- **Dans le code de la fonction il doit être assuré que chaque suite d'appels récursifs va toujours finir par atteindre un cas de base.**

Il est clair que le fait que la fonction s'arrête, signifie seulement qu'elle va fournir un résultat, mais non pas que ce résultat est correct. L'arrêt de la fonction est une condition préalable !

Dans notre exemple, il est donc nécessaire de préciser que la fonction « puissance » ne convient pas pour les exposants négatifs[7].

Dans une démonstration par récurrence en mathématiques la situation est semblable : Pour montrer qu'une propriété est vraie pour tout entier naturel, on montre d'abord qu'elle est vraie pour le cas de base n=0 et ensuite on montre que si la formule est vraie pour le naturel n, alors elle reste vraie pour n+1. La véracité de la propriété pour n=4 est alors ramenée successivement à n=3, n=2, n=1 jusqu'au cas de base n=0, que l'on sait vérifié.

---

[6] Dans cette situation le système risque d'entrer dans un état indéfini provoqué par un débordement de mémoire. Dans le pire cas il sera nécessaire de redémarrer l'ordinateur (RESET) avec toutes les conséquences que cela peut impliquer. Dans un cas plus favorable, Delphi détecte le problème et avorte l'exécution de la fonction.

[7] Elle ne convient pas non plus pour les exposants non entiers, mais ceux-ci sont de toute façon exclus vu que la variable exposant m est de type integer.

8.4 Fonctionnement interne

Dans ce paragraphe on va chercher une réponse à la question comment le système gère l'exécution d'une procédure récursive. La bonne compréhension de ce mécanisme est importante pour pouvoir rédiger des programmes efficaces.

Revenons à l'exemple de la fonction `factorielle` qui calcule la factorielle d'un nombre naturel donné. Cet exemple, déjà implémenté de façon itérative au chapitre précédent, se prête particulièrement bien à être programmé de façon récursive vu que la définition mathématique de la fonction se base sur des formules de récurrence.

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \; si \; n \in N_0 \end{cases} \text{ ou bien } \begin{cases} 0! = 1 \\ n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \; si \; n \in N_0 \end{cases}$$

```
function factorielle(n:integer):integer;
begin
  if n=0 then result:=1 else result:=n*factorielle(n-1)
end;
```

On peut l'intégrer dans le programme Delphi du chapitre précédent. Le reste du code reste identique.

Lors de l'exécution de `factorielle(1)` le système exécute la fonction jusqu'à l'appel récursif « `factorielle(n-1)` » A ce stade le système mémorise l'état actuel de toutes les variables locales de la fonction. Lors de l'exécution de « `factorielle(n-1)` » le système recommence à exécuter la fonction factorielle avec le paramètre `n-1=0`. Lors de ce deuxième passage dans la fonction, les variables locales sont réinitialisées, leurs anciennes valeurs ne sont pas accessibles à ce moment mais elles sont sauvegardées pour une réutilisation ultérieure. Maintenant on a donc `n=0` et le système termine ce passage dans la fonction avec `result:=1`. Ensuite le système revient dans le premier passage de la fonction à l'endroit `factorielle(n-1)`. Les variables locales récupèrent leur ancienne valeur et l'expression `n*factorielle(n-1)` est évaluée avec `n=1` et `factorielle(n-1)=1`. La variable `result` prend la valeur 1 et l'exécution quitte définitivement la fonction.

Pour des appels de fonction avec des arguments plus grands, par exemple `factorielle(15)` le système doit donc conserver autant de copies de toutes les variables locales qu'il y a d'appels récursifs avant d'atteindre le cas de base. Cela peut nécessiter une quantité appréciable de mémoire et de temps d'exécution pour la gestion qui en découle.

8.5 Exactitude d'un algorithme récursif

Il est souvent «facile » de montrer qu'une fonction récursive donne le bon résultat. Normalement le raisonnement par récurrence s'impose. Pour la fonction `factorielle`, par exemple :

Base : `n=0`, le résultat est effectivement 1.

Hypothèse de récurrence : supposons que la fonction `factorielle` donne le bon résultat jusqu'au rang `n-1` pour `n>0`, c'est-à-dire que `factorielle(n-1)=(n-1)!`.

Pont : montrons que la fonction donne encore le bon résultat au rang `n`, donc que `factorielle(n)=n!`. En effet, la fonction donne le résultat `n*factorielle(n-1)` qui est

égal à `n*(n-1)!` par l'hypothèse de récurrence et qui est encore égal à `n!` par la définition mathématique de la factorielle.

Cette démonstration ne prend pas en compte des problèmes de dépassement de mémoire, inhérents au système Delphi, si `n` est « trop grand ».

8.6 Comparaison : fonction récursive – fonction itérative[8]

L'exemple suivant est particulièrement impressionnant.

La célèbre suite de Fibonacci qui est définie par $\begin{cases} u_0 = 1 \; et \; u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \;, \; n \in N - \{0;1\} \end{cases}$, s'implémente directement par la fonction récursive suivante :

```
function fibo(n:integer): int64;
begin
 if n<=1 then result:=1
 else result:=fibo(n-1)+fibo(n-2)
end;
```

Le type du résultat est `int64`, une variante de `integer` qui permet de représenter des nombres entiers plus grands. La condition `n<=1` prend en charge les deux cas de base `n=0` et `n=1`.

La fonction suivante, dont l'algorithme est basé sur une simple boucle n'est pas vraiment difficile à comprendre non plus. Si `n` est différent de 0 et de 1, alors tous les termes de la suite jusqu'au n-ième sont calculés de proche en proche (comme on le ferait si on n'avait pas d'ordinateur à disposition).

```
function fibo_it(n:integer): int64;
var t0,t1: int64;
begin
 if n<=1 then result:=1
 else
  begin
   t0:=1;
   t1:=1;
   while n>1 do
     begin
      result:=t0+t1;
      t0:=t1;
      t1:=result;
      n:=n-1
     end
  end
end;
```

Pour se faire une idée de la situation, il est conseillé de tester les 2 fonctions. En prenant successivement les arguments 5, 10, 20, 30, 40, 50…, on va finir par constater la même chose indépendamment de l'ordinateur utilisé. Même sur un ordinateur rapide la fonction récursive va finir par devenir terriblement lente alors que la fonction itérative va rester rapide (résultat immédiat) même sur un ordinateur très faible ! Pourquoi ?

[8] Iteratif: algortihme basé sur une boucle.

Que fait la fonction itérative ? Si n>1, cette fonction effectue exactement n-1 additions sur des nombres du type int64, et un nombre approximativement proportionnel à n d'affectations, de soustractions et de comparaisons. Le temps d'exécution de ces opérations est négligeable par rapport à la croissance fulgurante des résultats : c'est-à-dire la fonction sera limitée par la capacité de représentation du type int64 avant qu'on remarque un quelconque ralentissement dans l'exécution !

Que fait la fonction récursive ? Etudions les exécutions de la fonction pour les arguments 2 à 5.

Pour fibo(2) [résultat : 2], la condition n'est pas vérifiée et la fonction calcule donc l'expression fibo(1)+fibo(0). Les deux appels récursifs donnent directement le résultat 1 et l'exécution se termine donc « assez rapidement » après seulement 3 passages (chaque fois 1 appel avec les arguments 0, 1 et 2) dans la fonction.

Pour fibo(3) [résultat : 3], la fonction calcule d'abord fibo(2)+fibo(1). Le premier appel nécessite trois passages (voir alinéa précédent) dans la fonction et le deuxième appel donne directement 1. Cela fait donc un total de 5 appels.

Pour fibo(4) [résultat : 5], la fonction calcule fibo(3)+fibo(2). Le nombre d'appels se calcule par 5 (pour fibo(3)) + 3 (pour fibo(2)) + 1 (pour fibo(4)) = 9.

Sans entrer dans tous les détails : Pour fibo(10) [résultat : 89], il faut 177 appels, pour fibo(20) [résultat : 10946], il faut 21891 appels et pour fibo(30) [résultat : 1346269], il faut 2692537 appels.

Il est clair que le nombre d'appels de fonctions doit être supérieur ou égal au résultat. En effet, les deux seuls cas de bases donnent le résultat 1 et l'appel récursif consiste à ajouter deux résultats intermédiaires. Le résultat final est donc atteint en ajoutant tant de fois le nombre 1. A côté du nombre impressionnant de calculs que la fonction effectue, il ne faut pas oublier la quantité d'espace mémoire nécessaire pour sauvegarder toutes les valeurs intermédiaires de la variable n.

8.7 Récursif ou itératif ?

Contrairement à ce que l'exemple précédent pourrait suggérer, un algorithme récursif n'est pas automatiquement « moins bon » qu'un algorithme itératif. Si on tient compte de l'évaluation interne d'une fonction récursive, on peut parfois trouver une variante efficace de la fonction récursive.
Voici par exemple une formulation récursive « efficace » de la fonction fibo. L'astuce consiste à utiliser une fonction auxiliaire récursive avec deux arguments supplémentaires, nécessaires pour passer les deux termes précédents à l'appel suivant. On remarquera que dans le code de cette fonction il n'y a qu'un seul appel récursif, de façon à ce que le nombre d'appels pour calculer le n-ième terme de la suite de Fibonacci ne dépasse pas n.

```
function fibo(n:integer):int64;
  function fib_aux(n:integer;i,j: int64): int64;
  begin
    if n<=1 then result:=i
    else result:=fib_aux(n-1,i+j,i)
  end;
begin
  result:=fib_aux(n,1,1)
end;
```

Un certain nombre de problèmes admettent une solution récursive « très élégante » (algorithme simple à rédiger et à comprendre, mais pas nécessairement efficace). Nous verrons des exemples de ce genre dans le chapitre sur les recherches et les tris[9].

Pour certains de ces problèmes, une formulation à l'aide de boucles est très compliquée. Mais une telle formulation existe toujours. Dans le pire des cas, il est nécessaire de « simuler » l'algorithme récursif en stockant toutes les valeurs intermédiaires des variables dans un tableau. C'est de cette façon-là que le programme Delphi lui-même évalue les fonctions récursives ; en effet le langage machine, seul langage compris par le processeur, est un langage relativement faible qui ne connaît pas le mécanisme de la récursivité.

8.8 Exercices

Exercice 8-1

a) Ecrivez un programme qui calcule la fonction d'Ackermann à deux variables naturelles définie par les égalités :

$$\begin{cases} Ack(0,m) = m+1 \\ Ack(k,0) = Ack(k-1,1) \ si \ k \geq 1 \\ Ack(k,m) = Ack(k-1, Ack(k,m-1)) \ si \ k,m \geq 1 \end{cases}$$

b) Essayez cette fonction pour quelques arguments. Par exemple `ack(3,5)=253`, `ack(4,0)=13`, `ack(4,1)=65533`,

Exercice 8-2

Complétez la fonction puissance pour qu'elle calcule aussi correctement des puissances à exposant entier négatif.

Exercice 8-3

Ecrivez une version récursive de la fonction puissance rapide.

Aide : utilisez les formules $\begin{cases} a^0 = 1 \\ a^{2n} = (a^2)^n \\ a^{2n+1} = a^{2n} \cdot a \end{cases}$

Expliquez pour quelles valeurs de `a` et de `n`, l'exécution va s'arrêter et donner un résultat correct.

Exercice 8-4

Ecrivez une version récursive de la fonction pgcd.

Exercice 8-5

Ecrivez un programme récursif qui transforme un nombre binaire en notation décimale.

Exercice 8-6

Ecrivez un programme récursif qui transforme un nombre décimal en notation binaire.

---

9 Voir par exemple « recherche dichotomique » et « quicksort ».

9 Comptage et fréquence

9.1 Algorithme de comptage

On veut réaliser une fonction qui compte le nombre d'occurrences d'une chaîne de caractères dans une liste donnée.

Pour réaliser cet algorithme on parcourt la liste élément après élément et on le compare avec la chaîne trouvée. Si les deux chaînes sont identiques, on augmente un compteur.

```
function compter(liste:TListbox;chaine:string):integer;
var
  i,r:integer;
begin
  r:=0;
  for i:=0 to liste.Items.Count-1 do
    if liste.Items[I] = chaine then r:=r+1;
  result:=r;
end;
```

Exercice 9-1

Réalisez une fonction qui permet de compter le nombre d'occurrences d'une lettre dans une chaîne de caractères donnée.

9.2 Fréquence d'une lettre dans une liste

Soit une liste dont les éléments sont des lettres. On veut savoir combien de fois chaque lettre est représentée dans la liste. Ce type d'algorithme est un élément très important des algorithmes de compression, comme par exemple celui de Huffman.

La solution proposée utilise de manière un peu inhabituelle les tableaux, mais de ce fait-là devient très élégante. On utilise comme indice du tableau des lettres.

Le résultat de l'analyse de la liste se trouve dans une deuxième liste passée comme paramètre et a le format suivant : `lettre : nb. d'occurrences`.

```
procedure frequence(liste:TListbox; var resultat:TListbox);
var
  i:integer;
  c:char;
  tableau:array['A'..'Z'] of integer;
  element:string;
begin
  for c:='A' to 'Z' do tableau[c]:=0;
  for i:=0 to liste.Items.Count-1 do
    begin
      element:=liste.Items[i];
      tableau[element[1]]:=tableau[element[1]]+1;
    end;
  resultat.Items.Clear;
  for c:='A' to 'Z' do
    resultat.Items.Append(c+' : '+inttostr(tableau[c]));
end;
```

Exercice 9-2

Réalisez un sous-programme qui permet de compter le nombre d'occurrences des différentes lettres dans une chaîne de caractères donnée. Le résultat du sous-programme est un tableau.

10 Recherche et tri

10.1 Introduction

Les algorithmes de recherche et de tri ont une très grande importance en informatique. C'est surtout dans le contexte des bases de données que ces algorithmes sont utilisés. Nous allons traiter en premier lieu les algorithmes de tri car pour fonctionner certains algorithmes de recherche présument des données triées.

Bien que l'illustration des différents algorithmes se base sur un petit nombre de données, il ne faut pas oublier que ces algorithmes sont en général appliqués sur un nombre très grand de données (plus que 10000, comme par exemple dans un annuaire téléphonique).

10.2 Sous-programmes utiles

Nous allons définir deux sous-programmes qui vont nous permettre de faciliter la programmation et le test des algorithmes de tri et de recherche.

10.2.1 Echange du contenu de deux variables entières

```
procedure echange (var liste:TListbox; posa,posb:integer);
var
  temp: string;
begin
  temp:=liste.Items[posa];
  liste.Items[posa]:=liste.Items[posb];
  liste.Items[posb]:=temp;
end;
```

Cette procédure échange le contenu de deux éléments d'une liste.

10.2.2 Remplissage d'une liste

```
procedure remplissage(var liste: TListbox; n: integer);
const
  a=ord('A');
var
  i,j:integer;
  s : string ;
begin
  liste.Clear;
  for i:= 1 to n do
   begin
    s:='';
    for j:=0 to random(5)+1 do
      s:=s+chr(random(26)+a);
    liste.Items.Append(s);
   end;
end;
```

Cette procédure remplit une liste avec `n` mots (comportant de 2 à 6 lettres) pris au hasard.

10.3 Les algorithmes de tri

Le but d’un algorithme de tri est d'arranger des données selon un critère imposé. Ce critère représente une relation entre les données comme par exemple le tri d’une liste de mots selon l’ordre lexicographique ou le tri d’une liste de nombre en ordre croissant.

10.3.1 Tri par sélection

10.3.1.1 Idée

On cherche dans la liste le plus petit élément et on l’échange avec le premier élément de la liste. Ensuite on recommence l’algorithme en ne prenant plus en considération les éléments déjà triés et ceci jusqu’à ce qu’on arrive à la fin de la liste.

10.3.1.2 Solution récursive

```
procedure tri_selection_r(var liste:TListbox; debut:integer);
var
  j,min:integer;
begin
  min:=debut;
  for j:=debut+1 to liste.Items.Count-1 do
    if liste.Items[j]<liste.Items[min] then min:=j;
  echange(liste,debut,min);
  if debut < liste.Items.Count-2 then
    tri_selection_r(liste,debut+1);
end;
```

10.3.1.3 Solution itérative

```
procedure tri_selection_i(var liste:TListbox);
var
  i,j,min:integer;
begin
  for i:=0 to liste.Items.Count-2 do
    begin
      min:=i;
      for j:=i+1 to liste.Items.Count-1 do
        if liste.Items[j]<liste.Items[min] then min:=j;
      echange(liste,i,min);
    end;
end;
```

10.3.1.4 Exercice

Exécutez pas à pas les deux algorithmes en utilisant la liste suivante :

E ;X ;E ;M ;P ;L ;E.

10.3.2 Tri par insertion

10.3.2.1 Idée

On considère un élément après l'autre dans la liste et on cherche sa position dans la liste déjà triée. Ensuite on l'insère à la juste position. De ce fait les éléments suivants de la liste doivent être déplacés.

10.3.2.2 Solution récursive

```
procedure tri_insertion_r(var liste: TListbox;gauche,droite: integer);
var j:integer;
    candidat:string;
    termine:boolean;
begin
  if gauche<droite then
   begin
    tri_insertion_r(liste,gauche,droite-1);
    candidat:=liste.Items[droite];
    j:=droite;
    termine:=false ;
    while (not termine) and (J>G) do
    begin
      liste.Items[j]:=liste.Items[j-1];
      j:=j-1;
    end
    else termine:=true ;
    if j<droite then
      liste.Items[j]:=candidat;
   end;
end;
```

10.3.2.3 Solution itérative

```
procedure tri_insertion_i(var liste:TListbox);
var  i,j:integer;
     candidat:string;
     termine: boolean ;
begin
  for i:= 1 to liste.Items.Count-1 do
    begin
      candidat:=liste.Items[i];
      j:=i;
      termine:=false;
      while (not termine) and (j>0) do
        if liste.Items[j-1] > candidat then
        begin
          liste.Items[j]:=liste.Items[j-1];
          j:=j-1;
        end
        else termine:=true;
      if j<i then
        liste.Items[j]:=candidat;
    end;
end;
```

10.3.2.4 Exercice

Exécutez pas à pas les deux algorithmes en utilisant la liste suivante :

C;A;R;T;O;O;N.

10.3.3 Tri rapide

10.3.3.1 Introduction

Le tri rapide (ang. : Quicksort) est l’algorithme de tri le plus utilisé. Il est réputé pour sa vitesse de tri qui pour des listes de grande taille est souvent bien supérieure à celle d’autres algorithmes de tri. Nous allons développer le tri rapide en plusieurs étapes.

10.3.3.2 Idée

L’algorithme de tri va diviser la liste en deux parties. Ces parties qui ne sont pas forcément de taille égale, sont alors triées séparément par le même algorithme. L’important dans cet algorithme est la stratégie comment la liste est divisée en deux sous-listes.

10.3.3.3 Développement d’une solution récursive

L’idée de base nous conduit à un algorithme récursif très simple :

```
procedure tri_rapide_r(var liste:TListbox;g,d:integer);
var i:integer;
begin
  if d>g then
    begin
      i:=division(liste,g,d);
      tri_rapide_r(liste,g,i-1);
      tri_rapide_r(liste,i+1,d);
    end;
end;
```

Les paramètres `g` et `d` limitent la partie de la liste qui doit être triée. Le premier appel de la procédure de tri a comme paramètres l’indice du premier et l’indice du dernier élément de la liste.

10.3.3.4 Division de la liste

Pour que cette procédure récursive fonctionne correctement il faut que la fonction division remplisse trois conditions :

1. L’élément avec l’indice i (indice qui est retourné comme résultat) se trouve à l’endroit définitif.
2. Tous les éléments à gauche de l’élément avec l’indice i sont plus petits ou égaux à celui-ci.
3. Tous les éléments à droite de l’élément avec l’indice i sont plus grands ou égaux à celui-ci.

Ces trois conditions nous amènent à une situation très agréable :

La première nous dit que l’élément avec l’indice i n’a plus besoin d’être déplacé.

La deuxième implique qu’aucun élément de la sous-liste gauche ne sera déplacé au-delà de l’élément avec l’indice i.

La troisième implique qu’aucun élément de la sous-liste droite ne sera déplacé avant l’élément avec l’indice i.

En plus on peut affirmer que les deux sous-listes obtenues de cette manière-ci peuvent être triées par la suite d’une manière indépendante.

La fonction division se présente de la manière suivante :

```
function division(var liste: TListbox;g,d:integer):integer;
var i,j : integer;
    candidat :string;
begin
  candidat:=liste.Items[d];
  j:=d-1; i:=g;
  while i<=j do
   begin
    if liste.Items[i]< candidat then i:=i+1
    else if liste.Items[j]> candidat then j:=j-1
    else begin echange(liste,i,j); i:=i+1 ; j:=j-1; end;
   end;
  echange(liste,i,d);
  result:=i;
end;
```

Pour commencer, on choisit l'élément qui se trouve à la fin de liste comme candidat. On va déterminer la position définitive du candidat dans la liste et s'arranger que tous les éléments de la liste qui se trouvent avant sont plus petits et tous ceux qui se trouvent après sont plus grands que le candidat.

La boucle parcourt la liste en commençant par le début (`if`) jusqu'au premier élément supérieur au candidat, ensuite (`else if`) elle parcourt la liste en commençant par la fin jusqu'au premier élément inférieur au candidat. Ces 2 éléments sont alors (`else`) échangés et on recommence.

A la fin de la boucle le candidat est mis à sa place définitive et l'indice de cette place est rendu comme résultat de la fonction. On vérifie facilement qu'au cas où le candidat se trouvait à sa bonne place (lorsqu'il est le plus grand élément de la liste), alors i=d et le dernier échange n'a aucun effet.

10.3.4 Tri par fusion

Cet algorithme ne figure pas au programme.

10.3.4.1 Idée

Si l'on dispose de deux listes triées, on peut assez facilement les fusionner[10] (mettre ensemble) afin d'obtenir une seule liste triée. Un tri par fusion utilisant cette approche peut être implémenté de façon récursive et de façon itérative. Les deux versions utilisent la même procédure `fusion`.

10.3.4.2 Fusion de deux liste triées

L'algorithme de fusion utilise un tableau auxiliaire de chaînes de caractères, de même taille que la liste à trier. Ce tableau peut être déclaré globalement ; cela évite de redéclarer le tableau à chaque appel de la procédure.

Les deux sous-listes à fusionner se trouvent de façon consécutive dans la liste. Les paramètres g et m sont les indices de début et de fin de la première sous-liste, alors que la seconde sous-liste commence à l'indice m+1 et se termine en d. Les deux sous-listes sont d'abord copiées dans le tableau auxiliaire : la 1re de façon croissante et la 2e de façon décroissante de sorte que les éléments les plus grands se trouvent au milieu. Ensuite la procédure commence à comparer les plus petits éléments

[10] La fusion est en quelque sorte une généralisation de l'insertion où l'on fusionne aussi deux listes triées, dont l'une n'est composée que d'un seul élément.

des deux-sous-listes (le plus à gauche et le plus à droite) et replace le plus petit des deux dans la liste principale et ainsi de suite.

```
procedure fusion(var liste: TListbox; g,m,d: integer);
var i,j,k : integer;
  tt : array[1..1000] of string; {mieux: déclarer tt globalement}
begin
  for i:=g to m do tt[i]:=liste.Items[i];
  for j:=m+1 to d do tt[d+m+1-j]:=liste.Items[j];
  i:=g;   j:=d;
  for k:=g to d do
   if tt[i]<tt[j]
    then begin liste.Items[k]:=tt[i]; i:=i+1 end
    else begin liste.Items[k]:=tt[j]; j:=j-1 end;
end;
```

10.3.4.3 Solution récursive

Il suffit de diviser la liste en deux, de trier séparément les deux parties et de fusionner les deux parties triées.

```
procedure tri_fusion_re(var liste: TListbox;g,d:integer);
var m:integer;
begin
 if g<d then
  begin
    m:=(g+d) div 2;
    tri_fusion_re(liste,g,m);
    tri_fusion_re(liste,m+1,d);
    fusion(liste,g,m,d);
  end;
end;
```

10.3.4.4 Solution itérative

La version itérative commence par considérer des sous-listes de longueur 1 qui sont ensuite fusionnées 2 à 2. Ensuite la liste n'est composée que de sous-listes de longueur 2 déjà triées. Celles-ci sont encore fusionnées 2 à 2 et ainsi de suite. D'étape en étape la longueur des sous-listes triées double (variable `step`). Toute la liste est triée lorsqu'il n'existe plus qu'une seule grande sous-liste.

```
procedure tri_fusion_it(var liste: TListbox);
var i,m,step:integer;
begin
 m:=liste.Items.Count;
 step:=1;
 i:=0;
 while step<m do
 begin
   while (i+2*step-1)<m do
   begin
     fusion(liste,i,i+step-1,i+2*step-1) ;
     i:=i+2*step;
   end;
   if (i+step)<=m then fusion(liste,i,i+step-1,m-1);
   {s'il reste une liste et une partie d'une 2e liste}
   step:=step*2;
```

```
   i:=0 ;
 end;
end;
```

10.4 Les algorithmes de recherche

Les algorithmes de recherche ont pour but de déterminer l'indice d'un élément d'une liste qui répond à un certain critère. Si l'on ne trouve pas l'élément, on retourne par convention –1. L'élément recherché encore appelé clé, peut figurer plusieurs fois dans la liste. On suppose que la liste n'est pas vide.

10.4.1 Recherche séquentielle

10.4.1.1 Idée

Il s'agit de l'algorithme de recherche le plus simple qui puisse exister : on commence à examiner la liste dès le début jusqu'à ce qu'on ait trouvé la clé. L'algorithme donne l'indice de la première occurrence de la clé. La liste ne doit pas être triée.

10.4.1.2 Solution itérative

```
function recherche_seq_i(liste:TListbox;cle:string):integer;
var
  i:integer;
  trouve:boolean;
begin
 i:=0;
 trouve:=false;
 while (not trouve) and (i<liste.Items.Count) do
   if liste.Items[i]=cle then
     trouve:=true
   else
     i:=i+1;
 if trouve then result:=i
 else result:=-1;
end;
```

10.4.2 Recherche dichotomique

10.4.2.1 Idée

La recherche dichotomique est très rapide et utilise une méthode bien connue : Si par exemple, vous devez chercher une localité dans l'annuaire téléphonique, vous ne commencez pas à la première page pour ensuite consulter une page après l'autre, mais vous ouvrez l'annuaire au milieu pour ensuite regarder si la localité se situe dans la première partie ou dans la deuxième partie de l'annuaire. Ensuite vous recommencez l'opération, c à d vous divisez de nouveau la partie contenant la localité recherchée en deux et ainsi de suite jusqu'à ce que vous ayez trouvé la localité. L'algorithme de la recherche dichotomique utilise le même procédé :

On divise la liste en deux parties. On regarde si la clé correspond à l'élément au milieu de la liste. Si c'est le cas on a terminé, sinon on détermine la partie qui contient la clé et on recommence la recherche dans la partie contenant la clé. **La liste ne doit pas forcément contenir la clé, mais elle doit être triée**.

10.4.2.2 Solution récursive

```
function dicho_r(liste:TListbox;cle:string;g,d:integer):integer;
var
  milieu:integer;
begin
  if g>d then result:=-1
  else
    begin
      milieu:=(g+d) div 2;
      if liste.Items[milieu] = cle then result:=milieu
      else if cle<liste.Items[milieu] then
        result:= dicho_r(liste,cle,g,milieu-1)
      else
        result:= dicho_r(liste,cle,milieu+1,d);
    end;
end;
```

10.4.2.3 Solution itérative

```
function dicho_i(liste:TListbox;cle:string):integer;
var
  milieu,g,d:integer;
begin
  g:=0;
  d:=liste.Items.Count-1;
  milieu:=(g+d) div 2;
  while (cle<>liste.Items[milieu]) and (g<=d) do
  begin
    milieu:=(g+d) div 2;
    if cle<liste.Items[milieu] then d:=milieu-1
    else g:=milieu+1;
  end;
  if cle=liste.Items[milieu] then result:=milieu
  else result:=-1;
end;
```

11 Opérations sur les polynômes

11.1 Introduction

Pour exécuter les opérations sur les polynômes de façon efficace, on crée un **nouveau type de variable structuré** appelé enregistrement (record). À l'intérieur d'un tel enregistrement on peut regrouper tous les éléments jugés importants pour les tâches à accomplir.

Dans le cas des opérations sur les polynômes les éléments importants à regrouper sont les **coefficients** et le **degré** du polynôme. Pour les coefficients on utilise une variable du type tableau (array) et pour le degré une variable du type entier (integer).

Ainsi on déclare le nouveau type de variable structuré pour les opérations sur les polynômes de la façon suivante:

```
type poly = record
  c:array[0..100] of extended;
  d:integer
end;
```

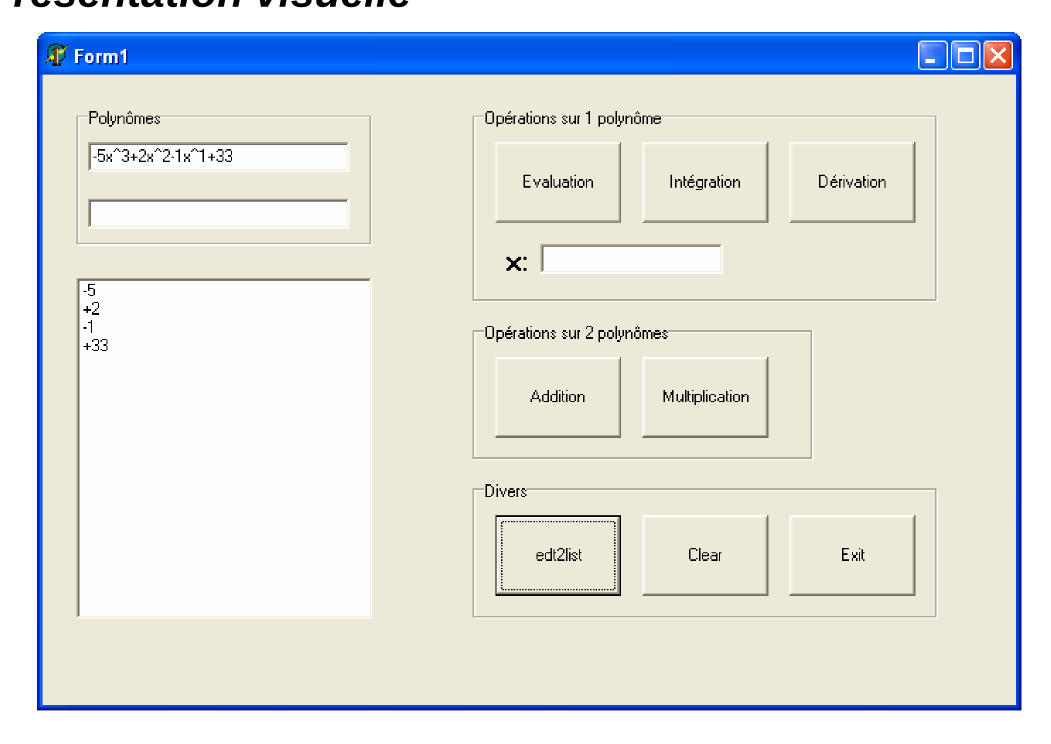
L'utilisation des sous-variables d'un enregistrement se fait par:

<nom de variable "enregistrement"> **.** <nom du champs>

Exemple:

```
var p:poly;
coeff:array[0..10] of extended;
degre,i:integer;
begin
  for i:=0 to 10 do
    coeff[i]:=p.c[i];
  degre:=p.d
```

11.2 Présentation visuelle



11.3 Transformation d’un polynôme donné sous forme de texte en une liste de coefficients

Il s’agit ici de transformer un polynôme donné sous forme de texte (TEdit, la notation est celle utilisée également dans Dérive/GeoGebra) en une liste de coefficients. La procédure `edt2list` est lancée par un clic sur le button « edtlist »

11.3.1 Fonctions/Procédures utiles:

La fonction `pos` sert à extraire des parties de chaînes de caractères. Sa syntaxe est la suivante :

```
pos(s1, s:string): integer;
```

où :

s1	sous-chaîne à rechercher dans la chaîne s ;
s	chaîne dans laquelle on recherce s1 ;

Si la chaîne `s` ne contient pas `s1`, la fonction retourne 0.

La procédure `delete` sert à effacer une partie d’une chaîne de caractères. Sa syntaxe est la suivante :

```
Delete(var s, index, count: Integer): string;
```

où :

S	chaîne de laquelle on veut effacer la partie ;
Index	indice de début de la partie à effacer;
Count	nombre de caractères à effacer.

11.3.2 Code

```
procedure edt2list(pol:string;var lbpoly:TListBox);
var coeff:string;
    pos1,pos2,posx:integer;

begin
  while (pos('x',pol)<>0) do
  begin
    //détermination de la place de l'exposant
    posx:=pos('x',pol);
    //détermination de l'endroit du prochain + ou -
    pos1:=pos('-',copy(pol,2,length(pol)-1));
    pos2:=pos('+',copy(pol,2,length(pol)-1));
    if ((pos1>pos2) and (pos2<>0)) or (pos1=0) then pos1:=pos2;

    //détermination du coefficient de l'exposant actuel
    coeff:=copy(pol,1,posx-1);
    lbpoly.Items.append(coeff);

    //effacement de la partie traîtée
    if pos1<>0 then delete(pol,1,pos1)
    else pol:='' // cas sans constante;
```

```
  end;

  if pol='' then lbpoly.Items.Append('0')
  else lbpoly.Items.Append(pol);                //constante
end;
```

11.3.3 Explications

Considérons le polynôme: -5x^3 + 2x^2 – 1x^1 + 33

Nous recherchons dans ce texte divers éléments à l'aide de la fonction *pos*

Recherche du coefficient actuel:

- *posx:* l'endroit où se trouve le prochain 'x' du texte
- *pos1:* l'endroit où se trouve le prochain '+' ou '–' du texte (on commence à chercher à partir de la 2ème lettre, car la première pourrait être un '–' comme dans l'exemple)

Nous recherchons *pos1* afin de pouvoir déterminer où se termine le monôme actuel

Ensuite, le coefficient actuel est extrait à l'aide de la fonction *copy* et ajouté à la liste. Ensuite le premier monôme est effacé de la chaîne de caractères à l'aide de la procédure *delete*.

Cette opération est répétée (while) jusqu'à ce qu'il n'y ait plus de 'x' dans la chaîne restante.Il suffit alors d'ajouter la chaîne restante (représentant la constante du polynôme) à la liste.

11.3.4 Améliorations

- acceptation de l'écriture x au lieu de x^1
- remplissage automatique par des '0' dans le cas d'un polynôme incomplet. Il faut donc à chaque fois vérifier l'exposant et comparer avec l'exposant précédent
- cas où le coefficient vaut +1 ou -1 et qu'on ne l'a pas écrit
- …

11.4 Transformation d'un polynôme donné sous forme de texte en une variable de type « poly »

Cette procédure sert à transformer un polynôme donné sous forme de texte (TEdit, la notation est celle utilisée également dans Dérive/GeoGebra) en une variable de type `poly`. La procédure `edt2poly` n'est pas appelée par un clic sur un button mais sera appelé par d'autres procédures lors des opérations sur les polynômes.

11.4.1 Code

```
procedure edt2poly(pol:string;var p:poly);
var coeff:string;
    pos1,pos2,posx,plexp,expo:integer;
begin
      //détermination de la place de l'exposant
      plexp:=pos('^',pol)+1;
```

```
      expo:=strtoint(copy(pol,plexp,pos1-plexp+1));
      p.d:=expo;

    while (pos('x',pol)<>0) do
    begin
      //détermination de l'endroit du prochain + ou -
      pos1:=pos('-',copy(pol,2,length(pol)-1));
      pos2:=pos('+',copy(pol,2,length(pol)-1));
      if ((pos1>pos2) and (pos2<>0)) or (pos1=0) then pos1:=pos2;

      //détermination du coefficient de l'exposant actuel
      posx:=pos('x',pol);
      coeff:=copy(pol,1,posx-1);
      p.c[expo]:=strtofloat(coeff);
      expo:=expo-1;

      //effacement de la partie traîtée
      if pos1<>0 then delete(pol,1,pos1)
      else pol:='' // cas sans constante;
    end;
    p.c[0]:=strtofloat(pol);                  //constante
end;
```

11.4.2 Explications

Le principe est le même que pour la liste de coefficients. Il suffit de déterminer le degré du polynôme (partie d), ensuite on détermine les différents coefficients (c[d] jusqu'à c[0]).

11.5 Transformation d'une variable de type « poly » en une liste de coefficients

Il suffit de parcourir la liste des coefficients (partie c[i]) à l'aide d'une boucle **for** et d'ajouter à chaque fois le nouveau coefficient à la liste.

11.5.1 Code

```
procedure poly2list(p:poly;var lbpoly:TListBox);
var i:integer;
begin
  for i:=p.d downto 0 do
    lbpoly.Items.append(floattostr(p.c[i]));
end;
```

11.6 Addition et multiplication de polynômes

11.6.1 Addition

```
function somme(a,b:poly):poly;
var i:integer ;
    s:poly;
begin
  if a.d<=b.d then
  begin
    s.d:=b.d;
```

```
    for i:=0 to a.d do
      s.c[i]:=a.c[i]+b.c[i];
    for i:=a.d+1 to b.d do
      s.c[i]:=b.c[i]
  end
  else
  begin
    s.d:=a.d;
    for i:=0 to b.d do
      s.c[i]:=a.c[i]+b.c[i];
    for i:=b.d+1 to a.d do
      s.c[i]:=a.c[i]
  end;
  while (s.d>0) and (s.c[s.d]=0) do
    s.d:=s.d-1;
  somme:=s
end;
```

11.6.2 Multiplication

```
function produit(a,b:poly):poly;
var i,j:integer;
    p:poly;
begin
  if ((a.d=0) and (a.c[0]=0)) or ((b.d=0) and (b.c[0]=0)) then
  begin
    p.d:=0;
    p.c[0]:=0
  end
  else
  begin
    p.d:=a.d+b.d;
    for i:=0 to p.d do
      p.c[i]:=0;
    for i:=0 to a.d do
      for j:=0 to b.d do
        p.c[i+j]:=p.c[i+j]+a.c[i]*b.c[j]
  end;
  produit:=p
end;
```

11.7 Dérivation et intégration de polynômes

11.7.1 Dérivation

```
function derivee(a:poly):poly;
var i:integer;
    der:poly;
begin
  for i:=1 to a.d do
    der.c[i-1]:=a.c[i]*i;
  if a.d=0 then
  begin
```

```
    der.d:=0;
    der.c[0]:=0
  end
  else
    der.d:=a.d-1;
  derivee:=der
end;
```

11.7.2 Intégration

La procédure donne la primitive à terme constant 0.

```
function primitive(a:poly):poly;
var i:integer;
    prim:poly;
begin
  prim.c[0]:=0;
  for i:=0 to a.d do
    prim.c[i+1]:=a.c[i]/(i+1);
  if (a.d=0) and (a.c[0]=0) then
    prim.d:=0
  else
    prim.d:=a.d+1;
  primitive:=prim
end;
```

11.8 Évaluation d'un polynôme

```
function horner(a:poly;x:extended):extended;
var i:integer;
    px:extended;
begin
  px:=a.c[a.d];
  for i:=a.d-1 downto 0 do
    px:=px*x+a.c[i];
  horner:=px
end;
```